50272-101

DTIC FILE COPY

REPORT DOCUMENTATION PAGE	1. REPORT NO. DCA/SW/MT-88/001j	2.	3. Recipient's Accession No.

4. Title and Subtitle
Defense Communications Agency
Upper Level Protocol Test System
TELNET Protocol Remote Driver Specification

5. Report Date
May 1988

6.

7. Author(s)

8. Performing Organization Rept. No.

9. Performing Organization Name and Address
Defense Communications Agency
Defense Communications Engineering Center
Code R640
1860 Wiehle Ave.
Reston, VA 22090-5500

10. Project/Task/Work Unit No.

11. Contract(C) or Grant(G) No.
(C)
(G)

12. Sponsoring Organization Name and Address

13. Type of Report & Period Covered
FINAL

14.

15. Supplementary Notes
For magnetic tape, see: ADA195128

16. Abstract (Limit: 200 words)

This document is part of a software package that provides the capability to conformance test the Department of Defense suite of upper level protocols including: Internet Protocol (IP) Mil-Std 1777, Transmission Control Protocol (TCP) Mil-Std 1778, File Transfer Protocol (FTP) Mil-Std 1780, Simple Mail Transfer Protocol (SMTP) Mil-Std 1781 and TELNET Protocol Mil-Std 1782.

17. Document Analysis a. Descriptors
Protocol Test Systems;
Conformance Testing;
Department of Defense Protocol Suite;

b. Identifiers/Open-Ended Terms
Internet Protocol (IP)
Transmission Control Protocol (TCP)
File Transfer Protocol (FTP)
Simple Mail Transfer Protocol (SMTP)
TELNET Protocol

c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited Release	UNCLASSIFIED	33
	20. Security Class (This Page) UNCLASSIFIED	22. Price

(See ANSI-Z39.18) See Instructions on Reverse OPTIONAL FORM 272 (4-77) (Formerly NTIS-35) Department of Commerce

DEFENSE COMMUNICATIONS AGENCY

UPPER LEVEL PROTOCOL TEST SYSTEM

TELNET PROTOCOL MIL-STD 1782
REMOTE DRIVER SPECIFICATION

MAY 1988

Disclaimer Concerning Warranty and Liability

This software product and documentation and all future updates to it are provided by the United States Government and the Defense Communications Agency (DCA) for the intended purpose of conducting conformance tests for the DoD suite of higher level protocols. DCA has performed a review and analysis of the product along with tests aimed at insuring the quality of the product, but does not warranty or make any claim as to the quality of this product. The product is provided "as is" without warranty of any kind, either expressed or implied. The user and any potential third parties accept the entire risk for the use, selection, quality, results, and performance of the product and updates. Should the product or updates prove to be defective, inadequate to perform the required tasks, or misrepresented, the resultant damage and any liability or expenses incurred as a result thereof must be borne by the user and/or any third parties involved, but not by the United States Government, including the Department of Commerce and/or The Defense Communications Agency and/or any of their employees or contractors.

Distribution and Copyright

Comments

Comments or questions about this software product and documentation can be addressed in writing to:

DCA Code R640
1860 Wiehle Ave
Reston, VA 22090-5500
ATTN: Protocol Test System Administrator

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. SCOPE AND PURPOSE

This specification describes the Protocol Test System and the program functions of the Telnet Remote Driver (RD). Section 2, "The Protocol Test System," summarizes the testing procedures used by the Protocol Test System. Section 3 on "Remote Driver Design and Protocol Testing" contains guidelines for developing a Telnet RD on a host where an implementation under test (IUT) resides. The role of the Telnet RD in protocol testing is also defined.

It should be noted that vendors of Telnet User IUTs need to provide a full service remote driver. When only a Server implementation is to be tested, the driver is reduced in scope. Appendix A describes this reduced driver.

2. THE PROTOCOL TEST SYSTEM

2.1 TEST TOOLS

The major components of the Protocol Test System are:

- The Test Scenario Language -- To enable a tester to specify standardized scripts;
- The Scenario Language Compiler and Libraries -- To produce code for automated testing;
- The Protocol Reference Implementation -- To define standard functions for the protocol being tested; and
- The Drivers -- Central, Surrogate, Slave, and Remote -- To execute tests and to provide communications links.

This test system causes an IUT to perform peer protocol exchanges with a reference implementation at the Protocol Test System site. To generate reproducible results, a script controls the driver that controls each IUT through its upper level interface. A complete test executes several scenarios of prescribed actions that exercise one or more protocol features.

2.2 GENERAL REQUIREMENTS OF THE CENTRAL, SURROGATE, SLAVE, AND REMOTE DRIVERS

The Central Test Driver (CTD), which coordinates and monitors protocol testing, combines the Surrogate and Remote implementation. The Surrogate Driver provides the transport mechanism between the CTD, the Slave, and the Remote Drivers.

The Slave Driver passes commands from the CTD to the protocol reference implementation, and sends back responses from the reference to the CTD. It also executes driver commands received from the CTD. In a similar manner, the Remote Driver executes driver commands from the CTD. Such commands, however, control the RD itself and do not affect the IUT's state, nor do they refer to IUT actions. The RD's primary role is communicating CTD protocol primitives and IUT responses over the network.

3. REMOTE DRIVER DESIGN AND PROTOCOL TESTING

The following sections describe the Remote Driver design and explain how all drivers interact to communicate protocol commands or responses in lab testing.

3.1 THE COMMAND CHANNEL

All drivers except the RD reside at the Protocol Test System site. The Remote Driver operates as a background process using a Transport protocol level connection, which links the RD at a remote site with the laboratory drivers and the reference and IUT protocols.

To set up the command channel, the RD sets up a passive listen on a well-known port and waits for the Surrogate Driver to perform an active open on that port. The command channel is ready when the Transport connection to the Remote and Slave Drivers has been established, and all drivers have initiated communications with their respective protocols. Figure 1 is a diagram of connection establishment, and Table 1 below gives destinations and port numbers.

Table 1. Destinations and Port Numbers

Destination	Port Number
RD Passive Open Port for User Testing	1202
IUT Server Telnet Passive Open Port	23
RD Passive Open Port for Server Testing	1203

3.2 FLOW OF COMMANDS

Packets containing protocol command codes travel over the command channel from the Central Driver, through the Surrogate Driver, to the Slave and Remote Drivers (Figure 2, and Table 3). The Slave and Remote Drivers translate these codes into the appropriate commands, then issue the commands. Similarly, responses from the reference and IUT protocols are received by the Slave and Remote Drivers and forwarded over the command channel to the Surrogate and Central Drivers. The Central Driver (CTD) determines from the combined test responses whether the IUT has functioned properly and reacts accordingly -- by taking the next appropriate action in the testing process -- and the flow of commands is repeated.

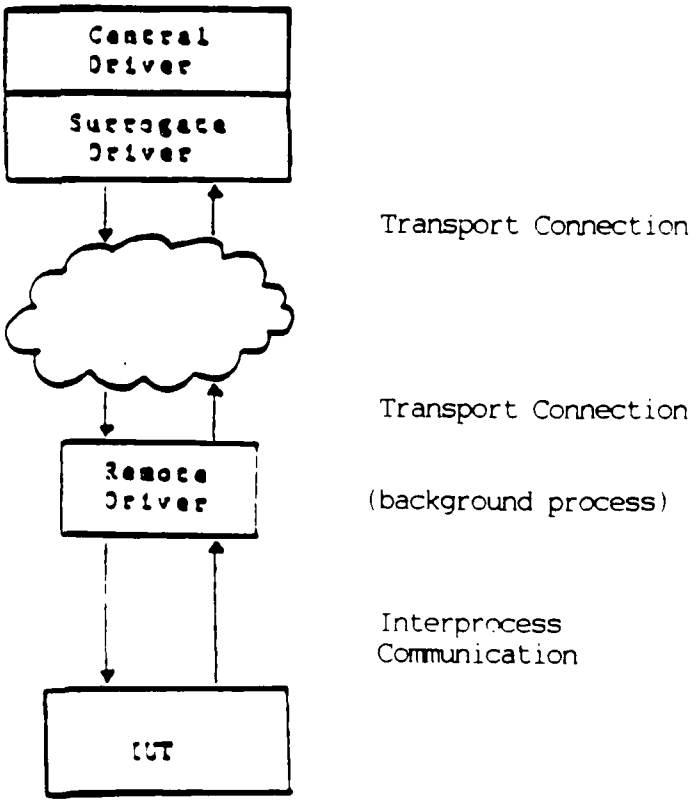

Figure 1. Connection Establishm

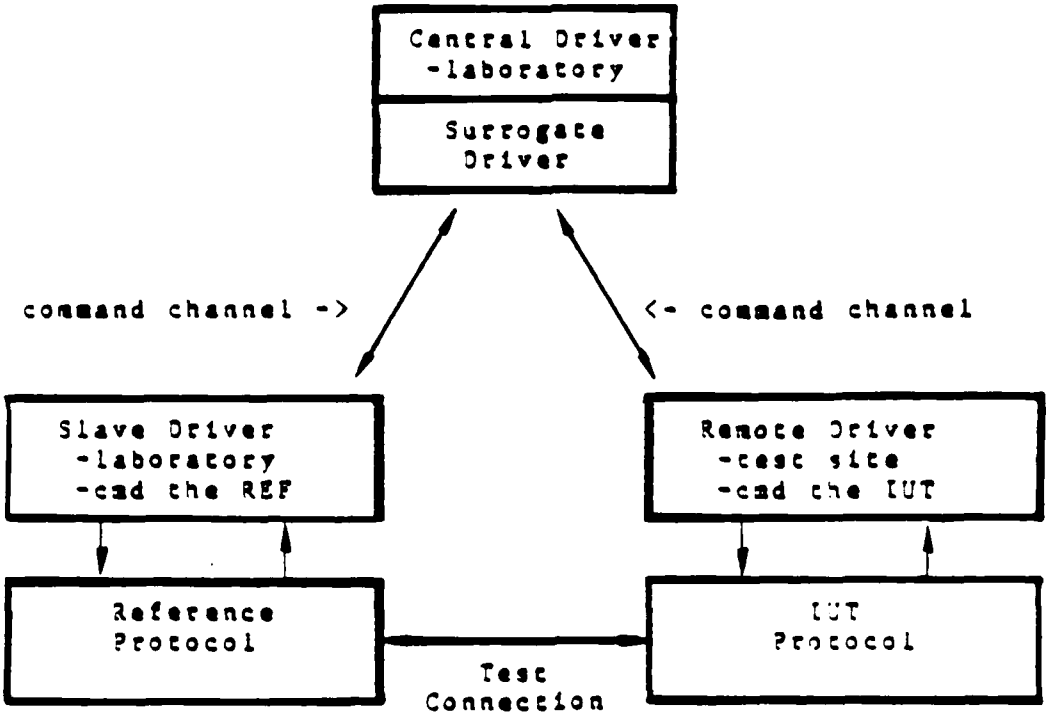

Figure 2. Flow of Commands Between the Four Types of Drivers

3.3 INPUTS AND OUTPUTS

Data is transmitted from the testing facility and to the DCA Laboratory in blocks called packets. The laboratory implementation of the Remote Driver uses a PAD (Packet Assembler/Disassembler) function to control the input and output of such packets.

The Remote Driver must be able to send one of three packets: an ACK (Acknowledgment), a NAK (Negative Acknowledgment), or a data packet containing either protocol responses or driver command results. Figure 3 shows the generic format of the data packet; Figure D.2 in Appendix D defines a data packet in C syntax.

3.3.1 The Control Flag Field

A single octet contains the control flag, which indicates by bit position how the Remote Driver should interpret the rest of the data packet. The bit positions are in ascending order from right to left, as in a Digital (Digital Equipment Corporation) VAX. Figure 4 diagrams the bit-ordering scheme.

Figure 3. Generic Format of the Data Packet

Control Flag:

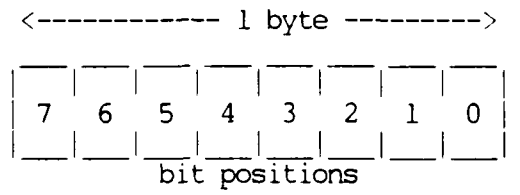

Figure 4. The Bit Order of a Byte

Table 2 summarizes the configuration of bit positions in the control flag. Only bit positions 0, 1, and 2 are used. The RD sets the bit in position 0 to one (1) to indicate that an ACK packet is being sent. A zero (0) in this position indicates a NAK. If the RD sets the bit in position 1, then a data packet (as opposed to an ACK/NAK packet) is being sent. Setting the DATA bit in a packet could signal either that the RD is sending a protocol response, or that it is sending the results of one of its own driver operations. The bit in position 2 of the first octet of each data packet sent by the CTD determines whether the packet contains a protocol or a driver command.

Table 2. Bit Positions in the Control Flag

Bit Position	Remote-->Central	Central-->Remote
0	ACK = 1 NAK = 0	unused
1	data = 1	unused
2	unused	protocol command = 1 driver command = 0
3-7	unused	unused

3.3.2 The Error Flag Field

Error flags will explain the nature of an error occurring at the Remote Driver. Because error codes are not yet implemented, the Protocol Test System makes no attempt to interpret this field.

3.3.3 The Primitive Code Field

The primitive code field can be interpreted in one of two ways -- either as a protocol primitive code or as a driver primitive code. A primitive in this sense means a command that describes some action within the protocol or the driver.

3.3.3.1 The Protocol Primitive Codes

If the control flag indicates a protocol command (i.e., the bit in position 2 is set to 1), then the Remote Driver must translate the integer in the code field to its corresponding protocol primitive according to the descriptions and the code numbers in Table 3. What form of data the RD sends to the protocol IUT depends on the IUT's User Interface, but the RD must include this translation capability in its function. To pass qualification testing, a Telnet IUT must be able to perform all the primitive actions described in Table 3.

The Remote Driver determines whether arguments are associated with a primitive according to its protocol IUT. If arguments are required for a given primitive, they will be supplied in the "data" field of the data packet. If such arguments are required but not supplied, then an error condition occurs and the RD must NAK that packet.

The RD can determine whether the required arguments are present by reading the "num_bytes" field, which tells how many bytes of character ASCII

data are in the "data" field. If this field contains a zero, then the RD assumes no arguments have been supplied. However, if the "num_bytes" field contains any positive integer from 0 to 4096, the RD must read the contents of the "data" field and interpret these contents as the primitive's arguments. An integer outside the range of 0 to 4096 means the packet is in error, and that a NAK packet should be returned.

In Table 3 the primitive commands and number codes are followed by one or more argument fields. Arguments are located in the data field of the command packet beginning at location zero and are separated by ASCII spaces. Three hyphens (---) in the table's "arguments" column indicate that no arguments exist.

Table 3. Protocol Primitive Commands, Number Codes, and Arguments, with Consequent Telnet IUT Action (Page 1 of 2)

Primitive (No. Code)	Argument	Telnet IUT Action
GA (0)	---	Causes the IUT to send a go-ahead command (IAC GA).
IP (1)	---	Causes the IUT to send an interrupt process command (IAC IP).
AO (2)	---	Causes the IUT to send an abort output command (IAC AO).
AYT (3)	---	Causes the IUT to send an Are-You-There command (IAC AYT).
EC (4)	---	Causes the IUT to send an erase character command (IAC EC).
EL (5)	---	Causes the IUT to send an erase line command (IAC EL).
SYNCH (6)	---	Causes the IUT to send a Telnet "synch" signal. (See MIL-STD-1782, p. 9.)
NOP (9)	---	Causes the IUT to send a no-op command (IAC NOP).
DO (10) DONT (11) WILL (12) WONT (13)	option option option option	Causes the IUT to send the indicated negotiation command. (Negotiation commands are DO, DONT, WILL, WONT.) The option negotiated is sent as an ASCII string as the first argument. These strings are indicated in Table 4.
STATUS (14)	---	Causes the IUT to request status information (if negotiated) from the reference implementation.

Table 3. Protocol Primitive Commands, Number Codes, and Arguments, with Consequent Telnet IUT Action (Page 2 of 2)

Primitive (No. Code)	Argument	Telnet IUT Action
OPEN (15)	host-name port-number	Requests the IUT to open a connection to the indicated host and port. The port number is optional. Both arguments are represented in ASCII form.
CLOSE (16)	---	Closes current session.
DM (17)	---	Causes the IUT to send a data mark command (IAC DM).
BRK (18)	---	Causes the IUT to send a break command (IAC BRK).

Table 4. Option Strings Sent As Data with Negotiation Commands

"BINARY"
"ECHO"
"SUPPRESS GO AHEAD"
"STATUS"
"TIMING MARK"

3.3.3.2 The Driver Primitive Codes

If the control flag indicates a driver command (the bit in position 2 is set to zero), the Remote Driver must translate the integer contained in the code field to its corresponding driver primitive (Table 5). The RD then performs the appropriate action, as specified in the following sections.

Table 5. The Driver Primitives

Code	Action
0	- Kill the Remote Driver process.
1	- Send associated data to the IUT.

3.3.3.2.1 The KILL Driver Primitive. If the Remote Driver interprets a driver primitive code of 0, then after ACKing the driver primitive packet, the process must find some way to terminate itself. No other action is necessary. The Remote Driver is not responsible for conducting a graceful shutdown of the protocol; it is the responsibility of the Central Driver to cause Telnet to quit. The RD also is not required to shut itself down gracefully, although this action is encouraged.

3.3.3.2.2 The DATA Driver Primitive. If the Remote Driver receives a driver primitive code of 1, indicating the DATA command, then after ACKing the driver primitive packet, the RD must send the text portion of the data packet to the IUT as data.

An octal 15 <015> in the data stream is to be interpreted as a carriage return. An octal 15 followed by an octal 12 <15><12>, is to be interpreted as a newline.

3.4 ACK/NAK PACKETS

The Remote Driver is required to acknowledge the receipt of every driver or protocol command (ACK = positive acknowledgment; NAK = negative acknowledgment). When it is able to read and interpret a packet from the TCP connection, the RD sends an ACK packet. If it detects an error, however, the Remote Driver responds with a NAK packet. The RD also sends a NAK packet if a read is successful but a code is unknown, or some other field is in error.

The following values indicate ACK or NAK packets:

ACK - code: 0

cntl_flag: bit position 0 set to one (1)
bit position 1 is unused
bit position 2 set to one (1) if protocol command,
to zero (0) if driver command

num_bytes: 0

data: empty

NAK - code: 0

cntl_flag: bit position 0 set to zero (0)
bit position 1 is unused
bit position 2 set to one (1) if protocol command,
to zero (0) if driver command

num_bytes: 0

data: empty

3.5 TIMING

A Remote Driver has no intrinsic timing constraints, but it should not add considerably to a protocol IUT's response time. For example, if the CTD does not receive a data packet within the time specified in a script, then a timeout condition will occur. Such a condition could cause an IUT to fail the test.

3.6 FLEXIBILITY

Although drivers have no special flexibility requirements, adaptable hardware and software enhances their operation and expandability. In the next phase of protocol testing, drivers may need to command passive recorders or other devices.

APPENDIX A
Remote Driver for Server Telnet IUT

In the case of Server Telnet implementations, the IUT vendor need not provide a full service IUT remote driver. What is required from the IUT vendor is a program that is executed by the reference User Telnet in the Server Telnet session. This program acts like a simplistic IUT driver in two respects: first, the Central Driver establishes a TCP connection with the program; second, the program handles a subset of commands the User Telnet IUT driver handles. This subset is limited to the driver commands handled by the User Telnet IUT driver. This program handles no protocol commands, and any protocol command received is NAKed immediately. Driver commands are ACKed or NAKed in the usual manner.

The program, which is called "telnetrsd," sets up a TCP passive connection on port 1203. A pseudo-code implementation follows:

```
begin telnetrsd
  reserve resources to listen on port 1203 for a TCP
    connection;
  print string "driver Telnet is ready";
  listen and wait on port 1203 for a TCP connection;
  forever do
  begin
    if a packet is received from the central driver then
      handle packet;
    if characters are received from the user Telnet then
      send the characters to the central driver in a
        data packet;
  end
end
```

Caveat: The Central Driver should be able to kill "telnetrsd" by sending it a "Kill Remote Driver" command.

APPENDIX B - References

"Military Standard File Transfer Protocol" (MIL-STD-1780); May 1984; Department of Defense.

"Military Standard TELNET Protocol" (MIL-STD-1782); August 1983; Department of Defense.

System Development Corporation, "Laboratory Implementation Plan," TM-WD-8574/000/02, January 1985.

System Development Corporation, "Laboratory Specification," TM-WD-7172/520/00, August 1984.

System Development Corporation, "Higher Level Capability Plan," TM-WD-8573/000/00, April 1984.

Kernighan, B. W., and Ritchie, D. M.; The C Programming Language; Prentice-Hall, Inc.; Englewood Cliffs, NJ; 1978.

Kernighan, B. W., and Pike, R.; The UNIX Programming Environment; Prentice-Hall, Inc.; Englewood Cliffs, NJ; 1984.

APPENDIX C - Glossary

ACK (Acknowledgment)

In data transfer between devices, data is blocked into units of a size given in each block's header. If the received data is found to be without errors, then the receiving device sends an ACK block back to the transmitting unit to acknowledge receipt. The transmitting device then sends the next block. If the receiving unit detects errors, however, it sends a NAK block to indicate the received data contained errors.

ASCII (American Standard Code for Information Interchange)

A standard code for the representation of alphanumeric information. ASCII is an 8-bit code in which 7 bits indicate the character represented and the 8th, high-order bit is used for parity.

CTD (Central Test Driver)

Digital (Digital Equipment Corporation)

IAC (Interpret As Command)

Telnet's escape charter, whose valve is 255 decimal.

IUT (Implementation Under Test)

A given vendor's protocol implementation and the subject of the immediate test.

MIL-STD (Military Standard)

Specification published by the Department of Defense.

NAK (Negative Acknowledgment)

In data transfer between devices, a NAK block is returned by the receiving device to the sending device to indicate the preceding data block contained errors. Also see ACK.

packet switching

A method of transmitting messages through a network in which long messages are subdivided into short packets. The packets are then transmitted as in message switching.

PAD (Packet Assembler/Disassembler)

In this document, PAD refers to a module of a structured program responsible for reading and writing data packets. Not to be confused with the other known usage, which describes a device to provide service to asynchronous terminals within an X.25 network.

PAR (Positive Acknowledgment and Response)

A simple communication protocol stating that every packet received must be either ACKed or NAKed.

protocol

A set of rules governing the operation of functional units to achieve communication.

TCP (Transmission Control Protocol)

The DoD standard connection-oriented transport protocol used to provide reliable, sequenced, end-to-end service.

Telnet

The DoD interim standard Virtual Terminal Protocol for the flexible attachment of remote terminals to host computer systems over the network.

APPENDIX D - Examples of Remote Driver Implementation in UNIX/C

D.1 CONNECTION ESTABLISHMENT

The Remote Driver must be able to establish interprocess communication; i.e., to read and write a data stream from a Transport (TCP) socket or from the Telnet IUT. Although the method of interprocess communication is not specified for the Remote Driver, a description of how a Remote Driver is implemented at the Protocol Test System site may be helpful. (See Figure D.1.)

The Protocol Test System runs in a UNIX 4.2 BSD environment. The Remote Driver is implemented in C language, which provides access to several interprocess communication system calls (e.g., fork, socket, bind, pipe, listen, and accept system calls).

Specifically, the IUT process is activated by the Remote Driver process using the fork system call. This forking causes a parent-child relationship to be formed between the two processes. Each process then determines whether it is the child or the parent. If the process is the parent, then it exits.

In effect, this action leaves the child process running as a background process. This background process then sets up the TCP socket connection by listening passively on the TCP port specified in Table 1.

After the TCP connection is established, the Remote Driver sets up communication with the IUT by forking another process and setting up inter-process communication through the use of the pipe system call. Once the pipes are established, the input/output of the two processes can be manipulated so that the two processes end up communicating with each other. Since the parent and child processes are exactly the same (except for process identification numbers) and would be useless talking to themselves, the protocol IUT is overlayed onto the child process using the exec system call.

When this procedure is successfully completed, the Remote Driver -- running as a background process -- is receiving and sending data over the TCP connection on one side, and sending/receiving data to/from the protocol IUT on the other side. (See Figure 1.)

D.2 A PAD FUNCTION

A Packet Assembler/Disassembler (PAD) function is useful for implementing a Remote Driver. Because the two basic functions of receiving and transmitting packets are performed repeatedly, it may be wise to modularize them. When the Remote Driver reads data from the command channel, it must be able to interpret the bytes correctly. In a UNIX/C implementation, the method used to achieve this result is to load the data into a data structure where distinct fields can be declared. In C, this is the "struct" declaration. Each collec-

tion of fields can be declared and referenced as a whole. The term "packet" has been used in this document as the name of this data structure reference. The "struct" declaration is shown in Figure D.2.

```
/*                  CONNECTION ESTABLISHMENT                                */
                              /* spawn IUT process, and                     */
                              /* put this process into background... */
                              /* parent returns pid; child returns 0 */
if ( (fork_stat = fork()) > 0)          /* pid > 0                           */
    exit();       /* so, if parent, then exit. */
                  /* else, this is child, so continue */

pp = getprotobyname("tcp");       /* 4.2 BSD system call */

while ((s = socket(AF_INET, SOCK_STREAM, pp->p_proto)) < 0)
{   if (errno != 0)
    {  perror("Telnet_SD:  socket");
       sleep(1);
    }
}
sin.sin_port = sp->s_port;  /* see port #                  */
                            /* bind name to socket        */
while (bind(s, (cadd_t)&sin, sizeof(sin), 0) < 0)
{   perror("Telnet_SD: bind");
    sleep(10);
}
listen(s, 10);     /* passive listen on socket */
                   /* accept connection        */
s2 = accept(s, (caddr_t)&from, &fromlen);
                   /* set up pipes             */

if (pipe(fd1) < 0)
   return(-1);
if (pipe(fd2) < 0)
   return(-1);
in_pipe = fd1[0];
out_pipe = fd2[1];
if (fork() == 0)  /* if child then execute the following */
     {  close (0);                          /* close std input            */
        if (dup2 (fd2[0], 0) < 0 )          /* open READ side of pipe     */
           perror ("dup2");
     close (1);                             /* close std output           */
     if (dup2 (fd1[1], 1) < 0)              /* open WRITE side            */
     execl ("/usr/iut/iut_telnet",  "iut_telnet", "-v", "-n", NULL);
     }
```

Figure D.1. Outline of Connection Establishment in 4.2 BSD UNIX/C

```
#define  MAX_TEXT_LEN       4096
struct remote_pack  {
              char cntl_flag;
              char err_flag;
              int  code;
              int  num_bytes;
              int  reserved;
              char text[MAX_TEXT_LEN];
            };
```

Figure D.2. The C Syntax Format of the Data Packet

The reception mode of the PAD function reads data and "packets" the data. The PAD accomplishes this task by reading the first 14 bytes of data from the input stream. This first 14 bytes is effectively the header of the data packet. It contains all the information needed to process the packet, including the integer in the "num_bytes" field that indicates the number of bytes of character text to follow, if any.

The transmission mode of the PAD function depackets the data and sends it over the communication channel. The PAD accomplishes this task by reading the header of the packet and writing the information into a memory space allocated for a large character string. The size is equal to 4096 bytes -- the maximum text size -- plus 14 bytes for the header information. If the text size is less than the maximum, then only that much need be written. The Remote Driver must transmit the data as a normal byte stream. If the input data stream is correctly packeted into a data structure, then interpretation of the fields in the packet should become clearer and less susceptible to error. An example of a PAD implemented in C is given in Figure D.3.

One important caveat: The following PAD routines are highly machine dependent in that they use the VAX word ordering convention when moving data from a read in packet to a C structure.

```
#define HEADER_SIZE   14
#define MAX_TEXT_LEN  4096
#define MAX_PACK_LEN HEADER_SIZE+MAX_TEXT_LEN

send_packet(packet,sock)      /* packet disassembler     */
struct remote_pack *packet;
int sock;
{    register int i;
     char send_buffer[MAX_PACK_LEN];

     send_buffer[0] = packet->cntl_flag;
     send_buffer[1] = packet->err_flag;
     *((int *)(send_buffer+2))    = packet->code;
     *((int *)(send_buffer+6))    = packet->num_bytes;
     *((int *)(send_buffer+10))   = packet->reserved;
     for(i=0;i<packet->num_bytes;i++)
          send_buffer[i+14] = packet->text[1];
                                  /* send the packet         */
     return(write(sock, send_buffer,
                  packet->num_bytes+HEADER_SIZE));
}

recv_packet(packet,sock)       /* packet assembler       */
struct remote_pack *packet;
int sock;
{
     char recv_buffer[HEADER_SIZE];
     int result;
                                /* read the header         */
     if   ( (result = read(sock, recv_buffer, HEADER_SIZE))  !=
                                                  HEADER_SIZE)
     {
          return (result);
     }
     packet->cntl_flag = recv_buffer[0];
     packet->err_flag  = recv_buffer[1];
     packet->code   = *((int *) (recv_buffer+2));
     packet->num_bytes = *((int *) (recv_buffer+6));
     packet->reserved  = *((int *) (recv_buffer+10));
                                /* read the text           */
     if (packet->num_bytes)
     {
       if(read(sock, packet->text, packet->num_bytes)  !=
                                          packet->num_bytes)
          return(-1);
     }
     return(0);
}
```

Figure D.3. A Packet Assembler/Disassembler in C